Bloom filter

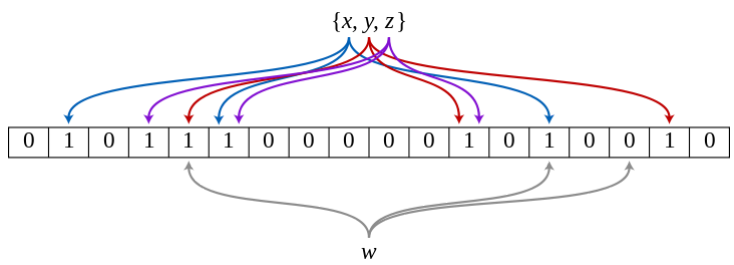
A **Bloom filter** is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter); the more elements that are added to the set, the larger the probability of false positives.

Bloom proposed the technique for applications where the amount of source data would require an impractically large amount of memory if "conventional" error-free hashing techniques were applied. He gave the example of a hyphenation algorithm for a dictionary of 500,000 words, out of which 90% follow simple hyphenation rules, but the remaining 10% require expensive disk accesses to retrieve specific hyphenation patterns. With sufficient core memory, an error-free hash could be used to eliminate all unnecessary disk accesses; on the other hand, with limited core memory, Bloom's technique uses a smaller hash area but still eliminates most unnecessary accesses. For example, a hash area only 15% of the size needed by an ideal error-free hash still eliminates 85% of the disk accesses.[1]

More generally, fewer than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set.[2]

Algorithm description

An *empty Bloom filter* is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions, generating a uniform random distribution. Typically, k is a constant, much smaller than m, which is proportional to the number of elements to be added; the precise choice of k and the constant of proportionality of m are determined by the intended false positive rate of the filter.



An example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit–array position containing 0. For this figure, $m = 18$ and $k = 3$.

To *add* an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.

To *query* for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem.

The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields. Alternatively, one can pass k different initial values (such as 0, 1, ..., $k - 1$) to a hash function that takes an initial value; or add (or append) these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in false positive rate.[3] (Specifically, Dillinger & Manolios (2004b) show the effectiveness of deriving the k indices using enhanced double hashing or triple hashing, variants of double hashing that are effectively simple random number generators seeded with the two or three hash values.)

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to k bits, and although setting any one of those k bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.

It is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

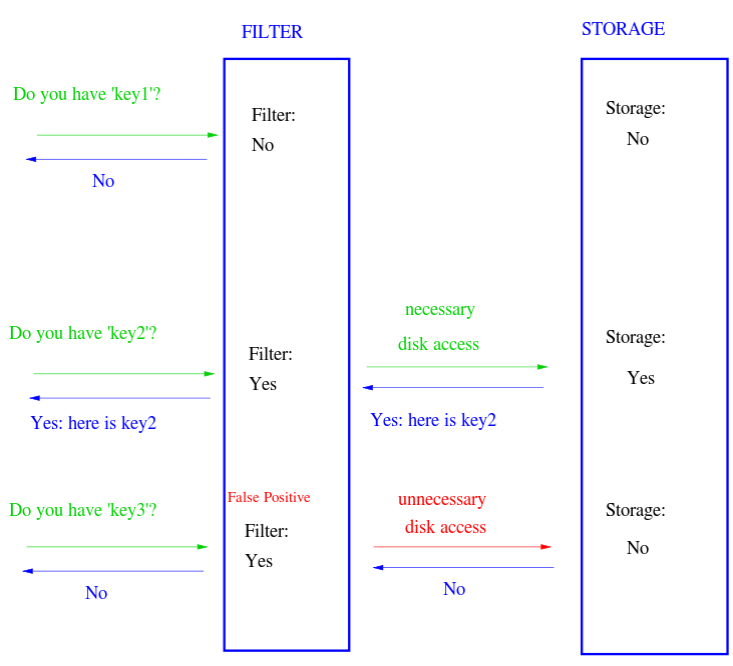
Space and time advantages

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). However, Bloom filters do not store the data items at all, and a separate solution must be provided for the actual storage. Linked structures incur an additional linear space overhead for pointers. A Bloom filter with 1% error and an optimal value of k, in contrast, requires only about 9.6 bits per element, regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. The 1% false-positive rate can be reduced by a factor of ten by adding only about 4.8 bits per element.

However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element. Note also that hash tables gain a space and time advantage if they begin ignoring collisions and store only whether each bucket contains an entry; in this case, they have effectively become Bloom filters with $k = 1$.[4]

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, O(k), completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its k lookups are independent and can be parallelized.

To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when $k = 1$. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter (k greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters (k and m) are chosen well, about half of the bits will be set,[5] and these will be apparently random, minimizing redundancy and maximizing information content.



Bloom filter used to speed up answers in a key–value storage system. Values are stored on a disk which has slow access times. Bloom filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a positive (in order to weed out the false positives). Overall answer speed is better with the Bloom filter than without the Bloom filter. Use of a Bloom filter for this purpose, however, does increase memory usage.

Probability of false positives

Assume that a hash function selects each array position with equal probability. If m is the number of bits in the array, the probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element is

$$1 - \frac{1}{m}.$$

If k is the number of hash functions and each has no significant correlation between each other, then the probability that the bit is not set to 1 by any of the hash functions is

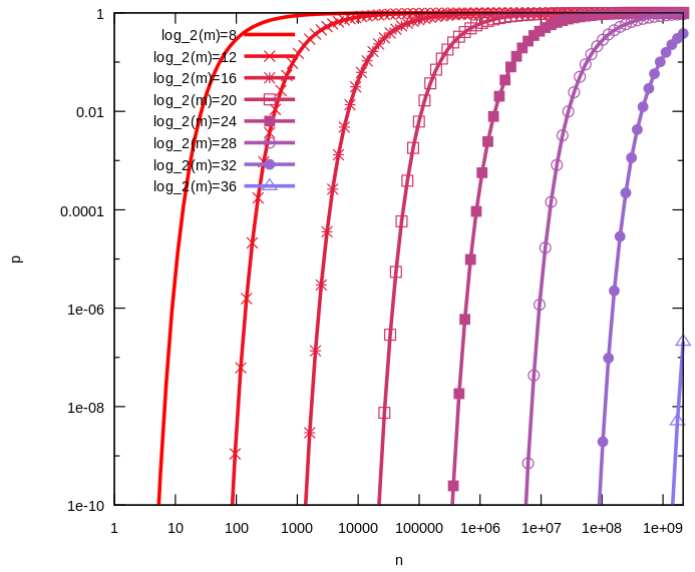
$$\left(1 - \frac{1}{m}\right)^k.$$

If we have inserted n elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn};$$

the probability that it is 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$



The false positive probability p as a function of number of elements n in the filter and the filter size m. An optimal number of hash functions $k = (m/n) \ln 2$ has been assumed.

Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is often given as

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

This is not strictly correct as it assumes independence for the probabilities of each bit being set. However, assuming it is a close approximation we have that the probability of false positives decreases as m (the number of bits in the array) increases, and increases as n (the number of inserted elements) increases.

An alternative analysis arriving at the same approximation without the assumption of independence is given by Mitzenmacher and Upfal.[6] After all n items have been added to the Bloom filter, let q be the fraction of the m bits that are set to 0. (That is, the number of bits still set to 0 is qm.) Then, when testing membership of an element not in the set, for the array position given by any of the k hash functions, the probability that the bit is found set to 1 is $1 - q$. So the probability that all k hash functions find their bit set to 1 is $(1 - q)^k$. Further, the expected value of q is the probability that a given array position is left untouched by each of the k hash functions for each of the n items, which is (as above)

$$E[q] = \left(1 - \frac{1}{m}\right)^{kn}.$$

It is possible to prove, without the independence assumption, that q is very strongly concentrated around its expected value. In particular, from the Azuma–Hoeffding inequality, they prove that[7]

$$\Pr(|q - E[q]| \geq \frac{\lambda}{m}) \leq 2\exp(-2\lambda^2/kn)$$

Because of this, we can say that the exact probability of false positives is

$$\sum_t \Pr(q = t)(1 - t)^k \approx (1 - E[q])^k = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

as before.

Optimal number of hash functions

The number of hash functions, k, must be a positive integer. Putting this constraint aside, for a given m and n, the value of k that minimizes the false positive probability is

$$k = \frac{m}{n} \ln 2.$$

The required number of bits, m, given n (the number of inserted elements) and a desired false positive probability p (and assuming the optimal value of k is used) can be computed by substituting the optimal value of k in the probability expression above:

$$p = \left(1 - e^{-(\frac{m}{n} \ln 2)\frac{n}{m}}\right)^{\frac{m}{n} \ln 2}$$

which can be simplified to:

$$\ln p = -\frac{m}{n}(\ln 2)^2.$$

This results in:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

So the optimal number of bits per element is

$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2} \approx -1.44 \log_2 p$$

with the corresponding number of hash functions k (ignoring integrality):

$$k = -\frac{\ln p}{\ln 2} = -\log_2 p.$$

This means that for a given false positive probability p, the length of a Bloom filter m is proportionate to the number of elements being filtered n and the required number of hash functions only depends on the target false positive probability p.[8]

The formula $m = -\frac{n \ln p}{(\ln 2)^2}$ is approximate for three reasons. First, and of least concern, it approximates $1 - \frac{1}{m}$ as $e^{-\frac{1}{m}}$, which is a good asymptotic approximation (i.e., which holds as $m \to \infty$). Second, of more concern, it assumes that during the membership test the event that one tested bit is set to 1 is independent of the event that any other tested bit is set to 1. Third, of most concern, it assumes that $k = \frac{m}{n} \ln 2$ is fortuitously integral.

Goel and Gupta,[9] however, give a rigorous upper bound that makes no approximations and requires no assumptions. They show that the false positive probability for a finite Bloom filter with m bits ($m > 1$), n elements, and k hash functions is at most

$$\left(1 - e^{-\frac{k(n+0.5)}{m-1}}\right)^k.$$

This bound can be interpreted as saying that the approximate formula $\left(1 - e^{-\frac{kn}{m}}\right)^k$ can be applied at a penalty of at most half an extra element and at most one fewer bit.

Approximating the number of items in a Bloom filter

Swamidass & Baldi (2007) showed that the number of items in a Bloom filter can be approximated with the following formula,

$$n^* = -\frac{m}{k} \ln\left[1 - \frac{X}{m}\right],$$

where n^* is an estimate of the number of items in the filter, m is the length (size) of the filter, k is the number of hash functions, and X is the number of bits set to one.

The union and intersection of sets

Bloom filters are a way of compactly representing a set of items. It is common to try to compute the size of the intersection or union between two sets. Bloom filters can be used to approximate the size of the intersection and union of two sets. Swamidass & Baldi (2007) showed that for two Bloom filters of length m, their counts, respectively can be estimated as

$$n(A^*) = -\frac{m}{k} \ln\left[1 - \frac{n(A)}{m}\right]$$

and

$$n(B^*) = -\frac{m}{k} \ln\left[1 - \frac{n(B)}{m}\right].$$

The size of their union can be estimated as

$$n(A^* \cup B^*) = -\frac{m}{k} \ln\left[1 - \frac{n(A \cup B)}{m}\right],$$

where $n(A \cup B)$ is the number of bits set to one in either of the two Bloom filters. Finally, the intersection can be estimated as

$$n(A^* \cap B^*) = n(A^*) + n(B^*) - n(A^* \cup B^*),$$

using the three formulas together.

Interesting properties

- Unlike a standard hash table using open addressing for collision resolution, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements; adding an element never fails due to the data structure "filling up". However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point *all* queries yield a positive result. With open addressing hashing, false positives are never produced, but performance steadily deteriorates until it approaches linear search.
- Union and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations respectively. The union operation on Bloom filters is lossless in the sense that the resulting Bloom filter is the same as the Bloom filter created from scratch using the union of the two sets. The intersect operation satisfies a weaker property: the false positive probability in the resulting Bloom filter is at most the false-positive probability in one of the constituent Bloom filters, but may be larger than the false positive probability in the Bloom filter created from scratch using the intersection of the two sets.
- Some kinds of superimposed code can be seen as a Bloom filter implemented with physical edge-notched cards. An example is Zatocoding, invented by Calvin Mooers in 1947, in which the set of categories associated with a piece of information is represented by notches on a card, with a random pattern of four notches for each category.

Examples

- The servers of Akamai Technologies, a content delivery provider, use Bloom filters to prevent "one-hit-wonders" from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.[10]

- Google Bigtable, Apache HBase and Apache Cassandra, and Postgresql[11] use Bloom filters to reduce the disk lookups for non–existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.[12]
- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).[13][14]
- The Squid Web Proxy Cache uses Bloom filters for cache digests (http://wiki.squid–cache.org/SquidFaq/CacheDigests).[15]
- Bitcoin uses Bloom filters to speed up wallet synchronization.[16]
- The Venti archival storage system uses Bloom filters to detect previously stored data.[17]
- The SPIN model checker uses Bloom filters to track the reachable state space for large verification problems.[18]
- The Cascading analytics framework uses Bloom filters to speed up asymmetric joins, where one of the joined data sets is significantly larger than the other (often called Bloom join in the database literature).[19]
- The Exim mail transfer agent (MTA) uses Bloom filters in its rate–limit feature.[20]
- Medium uses Bloom filters to avoid recommending articles a user has previously read.[21]
- Ethereum uses Bloom filters for quickly finding logs on the Ethereum blockchain.

Alternatives

Classic Bloom filters use $\mathbf{1.44\log_2(1/\epsilon)}$ bits of space per inserted key, where $\boldsymbol{\epsilon}$ is the false positive rate of the Bloom filter. However, the space that is strictly necessary for any data structure playing the same role as a Bloom filter is only $\mathbf{\log_2(1/\epsilon)}$ per key.[22] Hence Bloom filters use 44% more space than an equivalent optimal data structure. Instead, Pagh et al. provide an optimal-space data structure. Moreover, their data structure has constant locality of reference independent of the false positive rate, unlike Bloom filters, where a smaller false positive rate $\boldsymbol{\epsilon}$ leads to a greater number of memory accesses per query, $\mathbf{\log(1/\epsilon)}$. Also, it allows elements to be deleted without a space penalty, unlike Bloom filters. The same improved properties of optimal space usage, constant locality of reference, and the ability to delete elements are also provided by the cuckoo filter of Fan et al. (2014), an open source implementation of which is available.

Stern & Dill (1996) describe a probabilistic structure based on hash tables, hash compaction, which Dillinger & Manolios (2004b) identify as significantly more accurate than a Bloom filter when each is configured optimally. Dillinger and Manolios, however, point out that the reasonable accuracy of any given Bloom filter over a wide range of numbers of additions makes it attractive for probabilistic enumeration of state spaces of unknown size. Hash compaction is, therefore, attractive when the number of additions can be predicted accurately; however, despite being very fast in software, hash compaction is poorly suited for hardware because of worst-case linear access time.

Putze, Sanders & Singler (2007) have studied some variants of Bloom filters that are either faster or use less space than classic Bloom filters. The basic idea of the fast variant is to locate the k hash values associated with each key into one or two blocks having the same size as processor's memory cache blocks (usually 64 bytes). This will presumably improve performance by reducing the number of potential memory cache misses. The proposed variants have however the drawback of using about 32% more space than classic Bloom filters.

The space efficient variant relies on using a single hash function that generates for each key a value in the range $[0, n/\varepsilon]$ where $\boldsymbol{\epsilon}$ is the requested false positive rate. The sequence of values is then sorted and compressed using Golomb coding (or some other compression technique) to occupy a space close to $\boldsymbol{n\log_2(1/\epsilon)}$ bits. To query the Bloom filter for a given key, it will suffice to check if its corresponding value is stored in the Bloom filter. Decompressing the whole Bloom filter for each query would make this variant totally unusable. To overcome this problem the sequence of values is divided into small blocks of equal size that are compressed separately. At query time only half a block will need to be decompressed on average. Because of decompression overhead, this variant may be slower than classic Bloom filters but this may be compensated by the fact that a single hash function need to be computed.

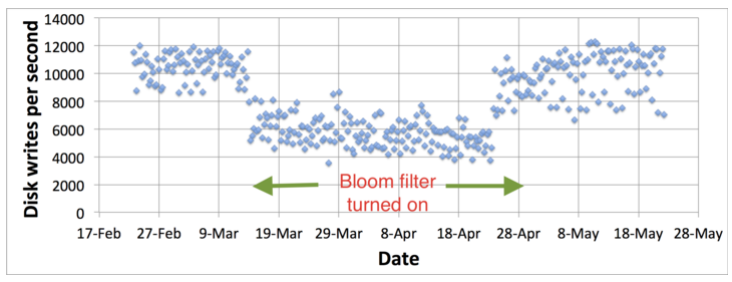
Another alternative to classic Bloom filter is the one based on space efficient variants of cuckoo hashing. In this case once the hash table is constructed, the keys stored in the hash table are replaced with short signatures of the keys. Those signatures are strings of bits computed using a hash function applied on the keys.

Extensions and applications

There are over 60 variants of Bloom filters, many surveys of the field, and a continuing churn of applications (see Luo, *et al*,[23] Dasgupta, *et al*[24]). Some of the variants differ sufficiently from the original proposal to be breaches from or forks of the original data structure and its philosophy.[25] A treatment which unifies Bloom filters with other work on random projections, compressive sensing, and locality sensitive hashing remains to be done.

Cache filtering

Content delivery networks deploy web caches around the world to cache and serve web content to users with greater performance and reliability. A key application of Bloom filters is their use in efficiently determining which web objects to store in these web caches. Nearly three-quarters of the URLs accessed from a typical web cache are "one-hit-wonders" that are accessed by users only once and never again. It is clearly wasteful of disk resources to store one-hit-wonders in a web cache, since they will never be accessed again. To prevent caching one-hit-wonders, a Bloom filter is used to keep track of all URLs that are accessed by users. A web object is cached only when it has been accessed at least once before, i.e., the object is cached on its second request. The use of a Bloom filter in this fashion significantly reduces the disk write workload, since one-hit-wonders are never written to the disk cache. Further, filtering out the one-hit-wonders also saves cache space on disk, increasing the cache hit rates.[10]



Using a Bloom filter to prevent one–hit–wonders from being stored in a web cache decreased the rate of disk writes by nearly one half, reducing the load on the disks and potentially increasing disk performance.[10]

Avoiding False Positives in a Finite Universe

Kiss *et al* [26] described a new construction for the Bloom filter that avoids false positives in addition to the typical non-existence of false negatives. The construction applies to a finite universe from which set elements are taken. It relies on existing non-adaptive combinatorial group testing scheme by Eppstein, Goodrich and Hirschberg. Unlike the typical Bloom filter, elements are hashed to a bit array through deterministic, fast and simple-to-calculate functions. The maximal set size for which false positives are completely avoided is a function of the universe size and is controlled by the amount of allocated memory.

Counting filters

Counting filters provide a way to implement a *delete* operation on a Bloom filter without recreating the filter afresh. In a counting filter the array positions (buckets) are extended from being a single bit to being an n-bit counter. In fact, regular Bloom filters can be considered as counting filters with a bucket size of one bit. Counting filters were introduced by Fan et al. (2000).

The insert operation is extended to *increment* the value of the buckets, and the lookup operation checks that each of the required buckets is non-zero. The delete operation then consists of decrementing the value of each of the respective buckets.

Arithmetic overflow of the buckets is a problem and the buckets should be sufficiently large to make this case rare. If it does occur then the increment and decrement operations must leave the bucket set to the maximum possible value in order to retain the properties of a Bloom filter.

The size of counters is usually 3 or 4 bits. Hence counting Bloom filters use 3 to 4 times more space than static Bloom filters. In contrast, the data structures of Pagh, Pagh & Rao (2005) and Fan et al. (2014) also allow deletions but use less space than a static Bloom filter.

Another issue with counting filters is limited scalability. Because the counting Bloom filter table cannot be expanded, the maximal number of keys to be stored simultaneously in the filter must be known in advance. Once the designed capacity of the table is exceeded, the false positive rate will grow rapidly as more keys are inserted.

Bonomi et al. (2006) introduced a data structure based on d-left hashing that is functionally equivalent but uses approximately half as much space as counting Bloom filters. The scalability issue does not occur in this data structure. Once the designed capacity is exceeded, the keys could be reinserted in a new hash table of double size.

The space efficient variant by Putze, Sanders & Singler (2007) could also be used to implement counting filters by supporting insertions and deletions.

Rottenstreich, Kanizo & Keslassy (2012) introduced a new general method based on variable increments that significantly improves the false positive probability of counting Bloom filters and their variants, while still supporting deletions. Unlike counting Bloom filters, at each element insertion, the hashed counters are incremented by a hashed variable increment instead of a unit increment. To query an element, the exact values of the counters are considered and not just their positiveness. If a sum represented by a counter value cannot be composed of the corresponding variable increment for the queried element, a negative answer can be returned to the query.

Decentralized aggregation

Bloom filters can be organized in distributed data structures to perform fully decentralized computations of aggregate functions. Decentralized aggregation makes collective measurements locally available in every node of a distributed network without involving a centralized computational entity for this purpose.[27]

Data synchronization

Bloom filters can be used for approximate data synchronization as in Byers et al. (2004). Counting Bloom filters can be used to approximate the number of differences between two sets and this approach is described in Agarwal & Trachtenberg (2006).

Bloomier filters

Chazelle et al. (2004) designed a generalization of Bloom filters that could associate a value with each element that had been inserted, implementing an associative array. Like Bloom filters, these structures achieve a small space overhead by accepting a small probability of false positives. In the case of "Bloomier filters", a *false positive* is defined as returning a result when the key is not in the map. The map will never return the wrong value for a key that *is* in the map.

Compact approximators

Boldi & Vigna (2005) proposed a lattice-based generalization of Bloom filters. A **compact approximator** associates to each key an element of a lattice (the standard Bloom filters being the case of the Boolean two-element lattice). Instead of a bit array, they have an array of lattice elements. When adding a new association between a key and an element of the lattice, they compute the maximum of the current contents of the k array locations associated to the key with the lattice element. When reading the value associated to a key, they compute the minimum of the values found in the k locations associated to the key. The resulting value approximates from above the original value.

Stable Bloom filters

Deng & Rafiei (2006) proposed Stable Bloom filters as a variant of Bloom filters for streaming data. The idea is that since there is no way to store the entire history of a stream (which can be infinite), Stable Bloom filters continuously evict stale information to make room for more recent elements. Since stale information is evicted, the Stable Bloom filter introduces false negatives, which do not appear in traditional Bloom filters. The authors show that a tight upper bound of false positive rates is guaranteed, and the method is superior to standard Bloom filters in terms of false positive rates and time efficiency when a small space and an acceptable false positive rate are given.

Scalable Bloom filters

Almeida et al. (2007) proposed a variant of Bloom filters that can adapt dynamically to the number of elements stored, while assuring a minimum false positive probability. The technique is based on sequences of standard Bloom filters with increasing capacity and tighter false positive probabilities, so as to ensure that a maximum false positive probability can be set beforehand, regardless of the number of elements to be inserted.

Spatial Bloom filters

Spatial Bloom filters (SBF) were originally proposed by Palmieri, Calderoni & Maio (2014) as a data structure designed to store location information, especially in the context of cryptographic protocols for location privacy. However, the main characteristic of SBFs is their ability to store **multiple sets** in a single data structure, which makes them suitable for a number of different application scenarios.[28] Membership of an element to a specific set can be queried, and the false positive probability depends on the set: the first sets to be entered into the filter during construction have higher false positive probabilities than sets entered at the end.[29] This property allows a prioritization of the sets, where sets containing more "important" elements can be preserved.

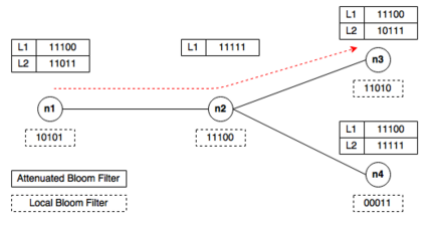
Layered Bloom filters

A layered Bloom filter consists of multiple Bloom filter layers. Layered Bloom filters allow keeping track of how many times an item was added to the Bloom filter by checking how many layers contain the item. With a layered Bloom filter a check operation will normally return the deepest layer number the item was found in.[30]

Attenuated Bloom filters

An attenuated Bloom filter of depth D can be viewed as an array of D normal Bloom filters. In the context of service discovery in a network, each node stores regular and attenuated Bloom filters locally. The regular or local Bloom filter indicates which services are offered by the node itself. The attenuated filter of level i indicates which services can be found on nodes that are i-hops away from the current node. The i-th value is constructed by taking a union of local Bloom filters for nodes i-hops away from the node.[31]

Let's take a small network shown on the graph below as an example. Say we are searching for a service A whose id hashes to bits 0,1, and 3 (pattern 11010). Let n1 node to be the starting point. First, we check whether service A is offered by n1 by checking its local filter. Since the patterns don't match, we check the attenuated Bloom filter in order to determine which node should be the next hop. We see that n2 doesn't offer service A but lies on the path to nodes that do. Hence, we move to n2 and repeat the same procedure. We quickly find that n3 offers the service, and hence the destination is located.[32]



Attenuated Bloom Filter Example: Search for pattern 11010, starting from node n1.

By using attenuated Bloom filters consisting of multiple layers, services at more than one hop distance can be discovered while avoiding saturation of the Bloom filter by attenuating (shifting out) bits set by sources further away.[31]

Chemical structure searching

Bloom filters are often used to search large chemical structure databases (see chemical similarity). In the simplest case, the elements added to the filter (called a fingerprint in this field) are just the atomic numbers present in the molecule, or a hash based on the atomic number of each atom and the number and type of its bonds. This case is too simple to be useful. More advanced filters also encode atom counts, larger substructure features like carboxyl groups, and graph properties like the number of rings. In hash-based fingerprints, a hash function based on atom and bond properties is used to turn a subgraph into a PRNG seed, and the first output values used to set bits in the Bloom filter.

Molecular fingerprints started in the late 1940s as way to search for chemical structures searched on punched cards. However, it wasn't until around 1990 that Daylight Chemical Information Systems, Inc. introduced a hash-based method to generate the bits, rather than use a precomputed table. Unlike the dictionary approach, the hash method can assign bits for substructures which hadn't previously been seen. In the early 1990s, the term "fingerprint" was considered different from "structural keys", but the term has since grown to encompass most molecular characteristics which can be used for a similarity comparison, including structural keys, sparse count fingerprints, and 3D fingerprints. Unlike Bloom filters, the Daylight hash method allows the number of bits assigned per feature to be a function of the feature size, but most implementations of Daylight-like fingerprints use a fixed number of bits per feature, which makes them a Bloom filter. The original Daylight fingerprints could be used for both similarity and screening purposes. Many other fingerprint types, like the popular ECFP2, can be used for similarity but not for screening because they include local environmental characteristics that introduce false negatives when used as a screen. Even if these are constructed with the same mechanism, these are not Bloom filters because they cannot be used to filter.

See also

- Count–min sketch
- Feature hashing
- MinHash
- Quotient filter
- Skip list